Department of Electronic and Telecommunication Engineering

University of Moratuwa

EN 3030 – Electronic Circuits and System Design

FPGA based Processor Design

A . W. U. M. Mendis	160405D
K. G. L. P. Nawarathne	160430A
N. L. Udugampola	160640R
N. P. Y. Yasawardena	160721R

This is submitted as a partial fulfillment for the module
EN3030: Electronic Circuits and System Design
Department of Electronic and Telecommunication Engineering
University of Moratuwa

25th July 2019

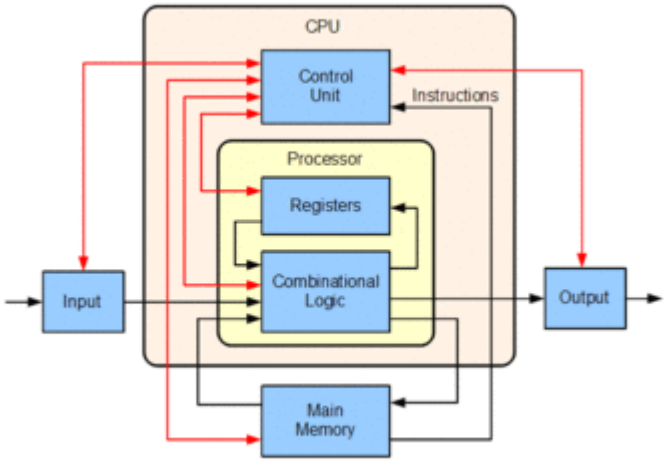
Contents.

Abstract

FPGA is a form of adaptable platform where any digital design can be programmed and configured. Processor designing can be deployed and tested with FPGAs in a more convenient manner. This report is about the task of implementing a fully automated custom processor. The designed processor is capable of down sampling a gray scale image of 255 x 255 pixels by a factor of two. This report includes algorithms, instruction set architecture(ISA), data paths and other resources used while designing the processor. As software packages, MATLAB was used whereas the hardware coding was done in Verilog HDL. Implementation was done in Quartus Prime 18.1 Light Edition along with Altera DE2-115 education and development board-Cyclone IV FPGA.

1. Introduction

1.1 Processor design overview



The above figure is a simple illustration of the architecture of a Central Processing Unit. CPU performs most of the processing inside the computer where the instruction set performs basic control, logical, input/output operations. To data flow and control instructions in the components inside the computer, the CPU relies heavily on a chipset located on the motherboard. As the processor takes a significant place in modern day digital design, it is of vital importance to design and implement a processor successfully. The processor we designed is not a general purpose one but a custom processor which can incorporate a specific application of downscaling an image by a factor of 2.

1.2 Problem Statement

Requirement- design a processor to downscale an image of 255x255 by a factor of 2 and display the result in the computer.

Step 01- converting the original image to mif file using Matlab code.

Then the mif file is loaded into the data memory of our design.

Step 02- assembly code for down sampling is written in a text document. by using a matlab code called compiler, we convert that text document to a mif file which contains the machine code. That mif file is loaded to the instruction memory. According to the instructions loaded to the instruction memory, image loaded in the data memory is processed.

Step 03- according to the algorithm described in section 4 of this document, image is Gaussian smoothed and down sampled by a factor of 2. These 2 happens at the same time in our designed processor. Down sampling images by selecting every other pixel can lead to a high probability of an error. Hence near pixel values are averaged and down sampled as a solution.

Step 04- after the down sampling process we read the 1st 127x127 pixels from the data memory using insistent memory access. We export those data as a hex file. Then that hex file is processed using a Matlab code to graphically store the down sampled image in our computer.

2. Instruction Set Architecture

The instruction set architecture is a detailed explanation of the processor design. This section covers the general architecture, modules used, how the buses are connected between modules and the data path. The algorithm is discussed in depth in section 4.

2.1 General Architecture

In section 3, registers will be deeply discussed, but for now the below table will help to get a general understanding about the modules and their architecture.

Register	**Size**	**Purpose**
P1-P9	8bit	Keeping the data memory address
R	16bit	Store the result of processing pixel intensity values

MAR	16bit	Keeping the memory address of the data memory word to read/write
PC	16bit	Keeping the memory address of the instruction to be loaded
PP	16bit	Keeping the memory address of the top left pixel of the current kernel
X	16bit	Register for keeping a pixel coordinate
Y	16bit	Register for keeping a pixel coordinate
G	16bit	Register for keeping a pixel coordinate
H	16bit	General purpose, directly connected to one ALU input

Instruction memory - Instruction memory can store 256 8 bit memories

Data memory-- Data RAM consists of 65536(256*256) memory locations having a width of 8 bits to store the pixels of the image

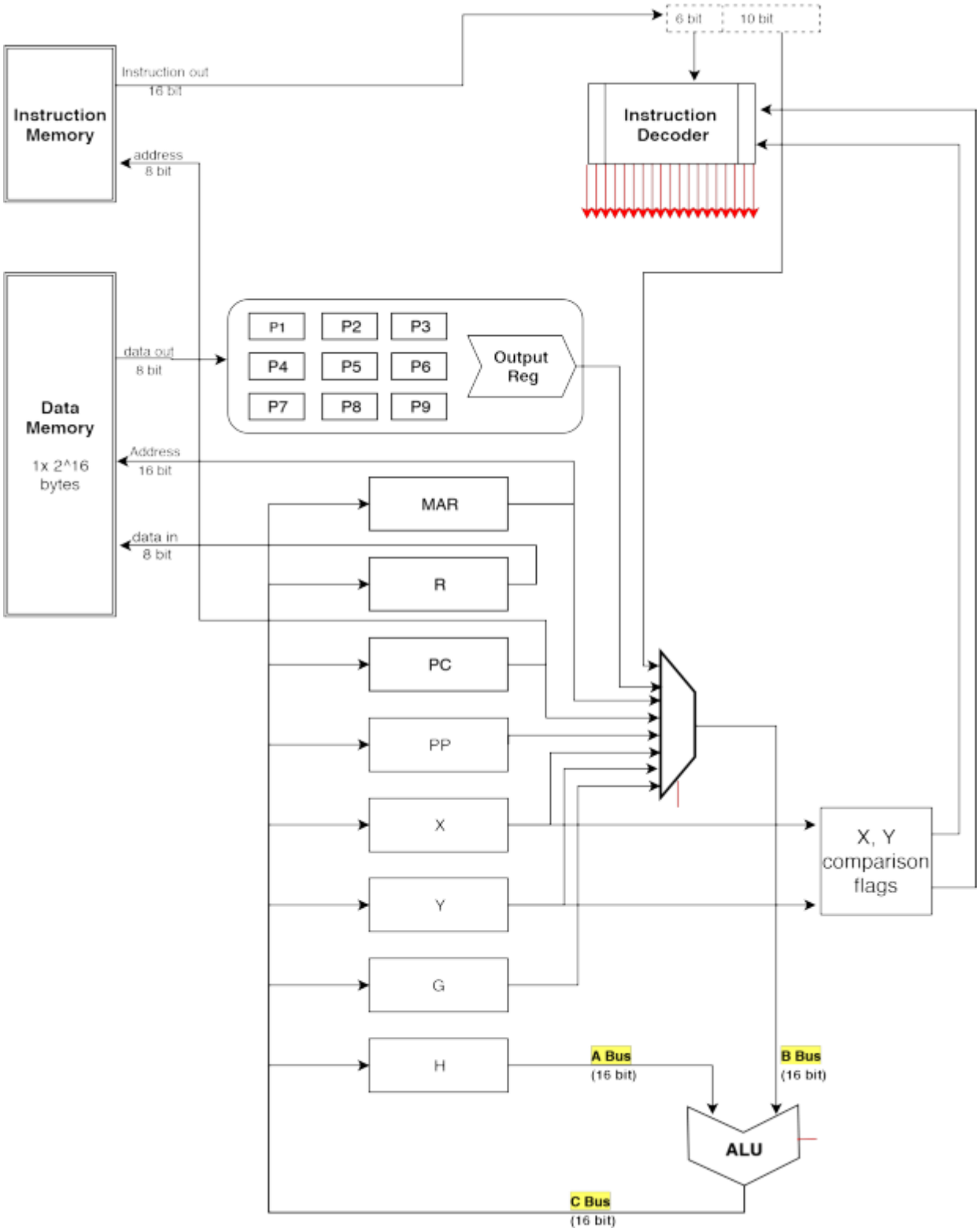
Instruction decoder- decodes the instruction and outputs a single set of control signals. Every instruction is executed in 1 clock cycle and there is no state machine in this architecture.

A bus - a 16 bit wire which is directly connected to H register

B bus - a 16 bit wire which carries data read from the registers(via a mux) to the ALU

C bus - a 16 bit wire writes data to the registers

2.2 Data Path

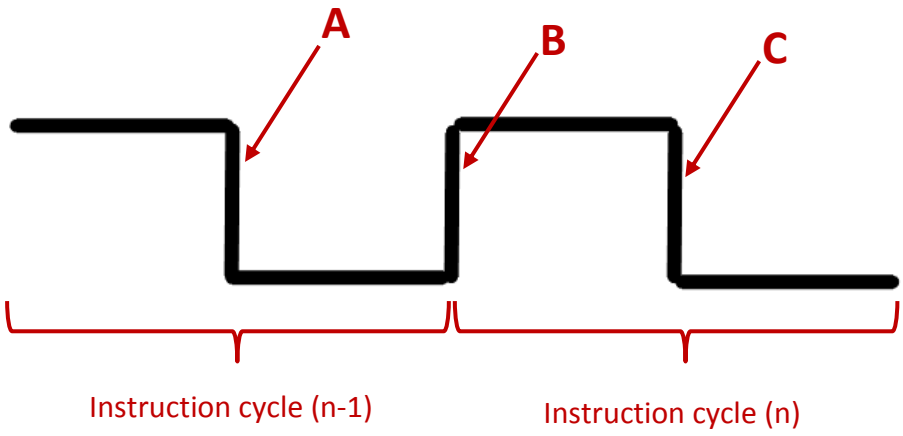

6 bit
10 bit
Instruction Memory
Instruction out
16 bit
address
8 bit
Instruction Decoder
Data Memory
1x 2^16 bytes
data out
8 bit
P1
P2
P3
P4
P5
P6
P7
P8
P9
Output Reg
Address
16 bit
data in
8 bit
MAR
R
PC
PP
X
Y
G
H
X, Y comparison flags
A Bus
(16 bit)
B Bus
(16 bit)
ALU
C Bus
(16 bit)

2.3 Instruction Set Architecture

Instruction	Instruction Code	Operation
NOOP	000001	No Operation
LDP1	000010	P1←M[MAR]
LDP2	000011	P2←M[MAR]
LDP3	000100	P3←M[MAR]
LDP4	000101	P4←M[MAR]
LDP5	011001	P5←M[MAR]
LDP6	011010	P6←M[MAR]
LDP7	011011	P7←M[MAR]
LDP8	011100	P8←M[MAR]
LDP9	011101	P9←M[MAR]
CAL	000110	R← (p1+p2*2+p3+p4*2+p5*4+p6*2+p7+p8*2+p9)/16
STR	000111	M[MAR]←R
PP2H	001000	H←PP
H2PP	001001	PP←H
ADD	001010 α	H←H+α
SUB	001011 α	H←H-α
ING1	001100	G←G+1
G2MAR	001101	MAR←G
H2MAR	010101	MAR←H
PP2MAR	010110	MAR←PP
CLH	001110	H←0
CLX	001111	X←0
CLY	010000	Y←0
INX	010001	X←X+1
INY	010010	Y←Y+1
JUMPX	010011 α	IF (X < preset value) THEN GOTO α
JUMPY	010100 α	IF (Y < preset value) THEN GOTO α
SETX	011110 α	X comparison value ← α
SETY	011111 α	Y comparison value ← α
STOP	b011000	End of program

2.4 Instruction Cycle

The custom processor that we designed has a RISC architecture. There's no state machine and every instruction is executed within one clock cycle. Here is a simple illustration on how this actually happens within one clock cycle. Let's a consider 2 consecutive clock cycles for simplicity.



Here we are mainly focusing on the second clock cycle where the n^{th} instruction cycle happens. Three clock edges directly contribute to everything happening in this n^{th} instruction cycle, including the negative clock edge A which belongs to the previous clock cycle. So let's start from there.

Negative clock edge A=>

Register PC holds the address of the next instruction to be executed. In this case the next instruction we are focusing on is the n^{th} instruction. So, in the negative edge A the PC register is updated with the instruction memory address of the next instruction. So in the negative clock edge A either PC gets automatically incremented by 1 or if the current (n-1) instruction tends to write a value in to PC register that value would get written. Following extraction from the Verilog module of PC register makes this more convenient.

```
always @ (negedge clk) begin
    if (pc_en) begin
        if (write_en) data_out<=data_in;
        else data_out<=data_out+1;
    end
end
```

Positive clock edge B=>

Now that the PC is successfully updated, its output register which is directly connected to the address input of the instruction memory keeps pointing at the next instruction to be fetched. At the positive edge B the instruction at that pointed address will get fetched in to the instruction decoder. Likewise, at the very beginning of the n^{th} instruction cycle the relevant instruction get fetched into the instruction decoder. Here onwards everything works like in a combinational circuit. As soon as the input opcode to the instruction decoder changes according to the newly fetched instruction, the set of control signals generated by the instruction decoder changes accordingly. These control signals may change the selection of

the register mux. Also, they may change the ALU operation. ALU also works as a combinational circuit behaving according to the selected setting by the control signals. With the inputs of the ALU and the selected control setting its output would get electrically stabilized and the C bus will have a stable value as well.

So, all these things happen during the high state of the clock after the positive edge B.

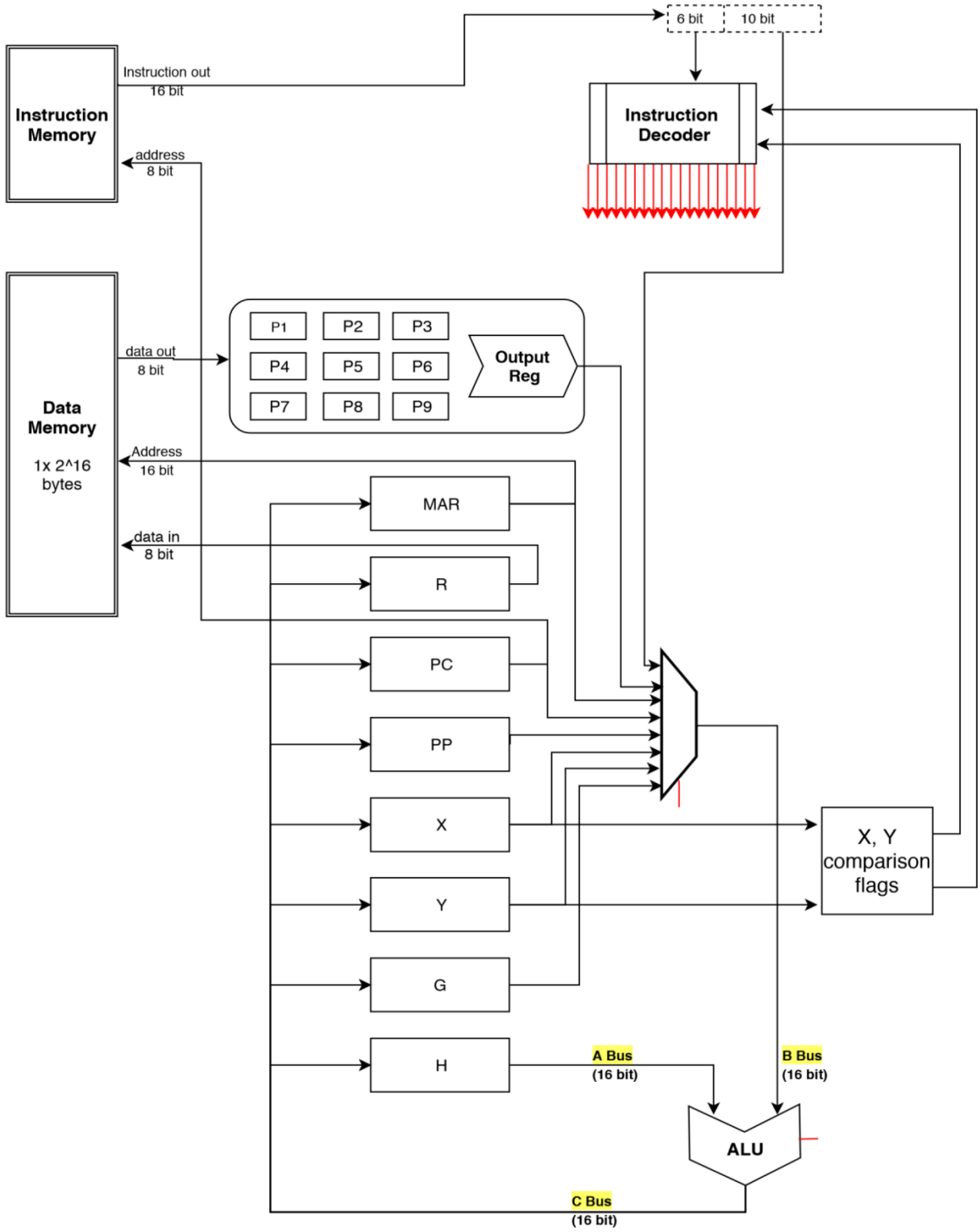
Negative clock edge C=>

Since everything has got stable by now, the C bus has got occupied with a stable value. So in the negative edge C, any data that is to be written into registers can be written from the C bus without any interference. On the other hand, just like in negative edge A, here the register PC will get updated with the address of the next (n+1) instruction. Following extraction from the Verilog module of a general register shows that writing into registers happens in this negative edge.

```
always @ (negedge clk) begin
    if (write_en) data_out<=data_in;
end
```

3. RTL Modules

The processor is a combination several inter connected hardware modules as shown in the below block diagram.



The processer has two hierarchical levels. In the lower hieratical level each individual hardware module is implemented using Verilog in the behavioral manner. In the upper

hierarchical level, the instances of the individual hardware modules are interconnected using wires and busses in the structural manner.

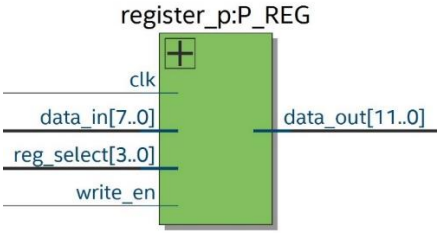
List of hardware modules.

3 Registers
4 Register multiplexer
5 Arithmetic and Logic Unit
6 X, Y comparison flag generator
7 Instruction Decoder
8 Instruction memory
9 Data memory

3.1 Registers

The processer has 17 registers in total. All the registers are 16 bit wide. The nine P registers (P1 to P9) and PC has its own behavioral model. The rest is generated from the same general purpose register model.

3.1.1 P register module



Input:

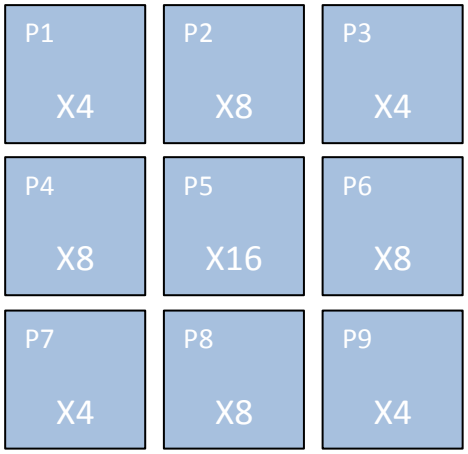
Clk	main clock input
data_in	Data input to the P registers. 8 bit wide.
reg_select	Selects the currently active register out of nine P registers. 4 bits wide.
write_en	write enable for all the P registers.

Output:

data_out	data output from the module. 12 bit wide.

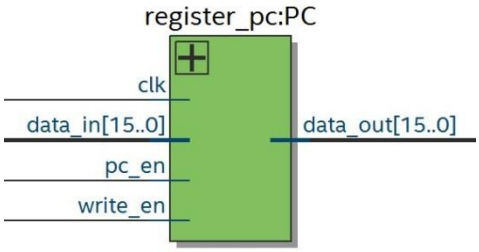
The P register module takes inputs from the Data memory where the data of the original image is stored. Since there are 9 general purpose registers inside, the user can load the values of 9 different pixels consecutively. The processer uses a 3x3 Gaussian kernel to filter the high frequencies from the original image before the down sampling process. The Gaussian kernel is implemented in hardware as a combinational logic circuit using the P1 to P9 registers. Therefore, the output of the P register module directly produces the Gaussian

filtered value of the center pixel (P5) of the 3x3 Gaussian kernel. The hardware implementation of the 3x3 Gaussian filter significantly reduce the number of clock cycles needed for the filtering process.



data_out = P1<<2+P2<<3+P3<<2+P4<<3+P5<<4+P6<<2+P7<<2+P8<<3+P9<<2

3.1.2 Program Counter (PC)



Input:

Clk	main clock input
data_in	Data input to the PC. 16 bit wide.
pc_en	Enables the functionality of PC
write_en	if (write_en= 1) then data_out=data_in else data_out=data_out+1

Output:

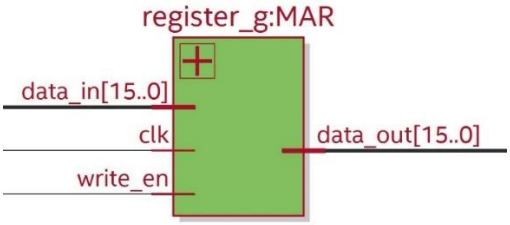
data_out	data output from the module. 16 bit wide.

The Program Counter Holds the Memory address of the next instruction to be executed. All the instructions for the processer are single clock cycle executable. (no microinstructions)

Therefore, under the normal operation (write_en is low) PC is incremented by one at every falling edge of the main clock. When write_en is high, data appears at data_in will be overwrite the existing value of PC.

3.1.3 General purpose registers

All the registers except P registers and PC are generated from the same general purpose register module.



Input:

Clk	main clock input
data_in	Data input to the register. 16 bit wide.
write_en	If (write_en=1) then data_out=data_in Else no change

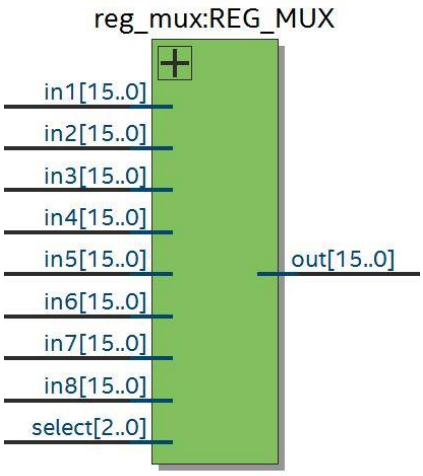
Output:

data_out	Data stored in the register. 16 bit wide.

The basic functionality of a general purpose register is to store and hold the data appearing at the data_in at every falling edge of the main clock if the write_en is high. All the general purpose registers are user accessible. The assigned task of every general purpose register is given below.

MAR	Memory Address Register This register holds the memory address of data memory
R	This register is used to hold the data which is to be written to the data memory in the following clock cycle.
PP	Pixel Pointer This register holds the memory address of the current pixel of the original image which is been processed.
X	Holds the x coordinate of the current pixel pointed by PP
Y	Holds the y coordinate of the current pixel pointed by PP
H	Store the result of the arithmetic operations done by the ALU
G	No specific task. Can be used to store temporary values.

3.2 Register Multiplexer



Input:

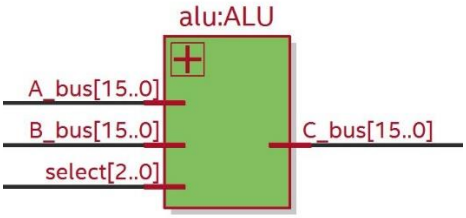
in1 to in8	16 –bit Data inputs to the multiplexer.
Select	3-bit selection input. Select the active input which will be connected to the output.

Output:

Out	Data out of the multiplexer. 16 bit wide.

The purpose of this module is to connect the general purpose registers (except R and H) and the immediate data from the instructions (last 10 bits) to the B_bus of the ALU one at a time. The select is used to select the active input connection. This module is implemented as a combinational logic circuit.

3.3 Arithmetic and Logic Unit (ALU)



Input:

A_bus	16 bit wide data input.
B_bus	16 bit wide data input.
select	3-bit selection input. Selects the arithmetic operation between A_bus and B_bus.

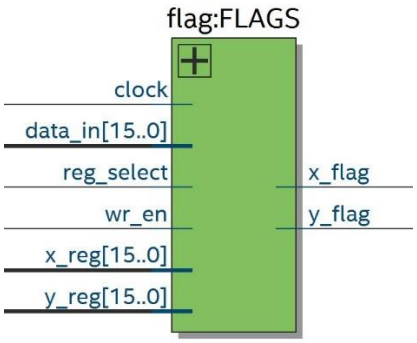
Output:

C_bus	16 bit wide data output.

This module performs simple arithmetic operations between the inputs A_bus and B_bus and output the result in C_bus. This is a combinational circuit. Currently no logical operation is performed by the ALU since it is not required for image down sampling. Available arithmetic operations are given below.

select	Operation
000	C_bus = A_bus
001	C_bus = B_bus
010	C_bus = 0
011	C_bus = A_bus + 1
100	C_bus = A_bus + 2
101	C_bus = B_bus + 1
110	C_bus = A_bus + B_bus
111	C_bus = A_bus - B_bus

3.4 X, Y comparison flag generator



Input:

clock	Main input clock
data_in	16 bit wide data input.
Reg_select	Selects the register which the data will be written to.
wr_en	Write enable for the internal registers
X_reg	Data input from the X register
Y_reg	Data input from the Y register

Output:

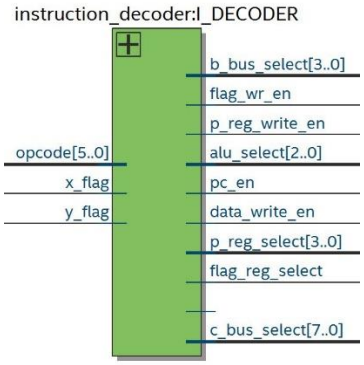
X_flag	1 bit wide data output.
Y_flag	1 bit wide data output.

Internal registers:

X_comp	16 bit register
Y_comp	16 bit register

The purpose of this module is to provide the logical data which is required for the branching instructions such as JUMP, JUMPX and JUMPY. The module takes inputs directly from the outputs of X and Y registers and compare it with the values in X_comp and Y_comp registers respectively. X_flag is driven high if the value of X register is less than the value if the X_comp register and wise versa. The user can set the values of X_comp and Y_comp registers at any time using the SETX and SETY instructions.

3.5 Instruction Decoder



Input:

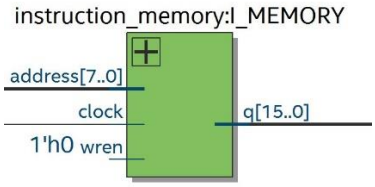
opcode	6-bit data input. Connected to the fist 6 bits of the instructions bus.
x_flag	1-bit data input. Used for branching instructions
y_flag	1-bit data input. Used for branching instructions

Output:

b_bus_select	4-bit control signal for the register multiplexer module.
flag_wr_en	write enable control signal for the flag generating module
p_reg_write_en	Write enable control signal for the P register module.
alu_select	3-bit control signal for the ALU
pc_en	Control signal for PC register. The processer can be start and stop using this signal.
data_write_en	Write enable signal for the data memory module
p_reg_select	4-bit selection control signal for the P register module
flag_reg_select	2-bit selection control signal for the flag generating module.
C_bus_select	8 bit control signal which acts as the write enable signal for the 8 general purpose registers.

The purpose of this module is to generate the corresponding control signals for a specific instruction by decoding the first 6 bits of the instruction (OP CODE). At each rising edge of the main clock, a new instruction is fetched from the instruction memory. Since the Instruction decoder is a combinational logic circuit, the control signals are generated immediately after the instruction fetching is completed.

3.6 Instruction memory



Input:

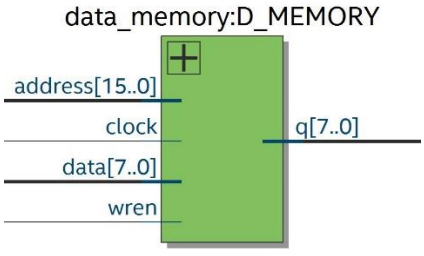
clock	Main clock input
address	8-bit data input which provides the memory address
wren	Write enable signal. Hard wired to logic 0.

Output:

q	16 bit data output.

This module is a 16 x 256 ROM generated from the in-built 1-port memory IP core of the QUARTUS Prime IDE. It is used to hold the user program inside the processer. The program is loaded to the Instruction memory using a memory initialization file (.mif) during the compilation process. The content inside the instruction memory cannot be edited during the execution of the program. At each rising edge of the main clock, the output q gives the data stored in the memory location pointed by the input address.

3.7 Data memory



Input:

clock	Main clock input
address	16-bit data input which provides the memory address
wren	Write enable signal
data	8-bit data input

Output:

q	8 bit data output

This module is an 8 x 65536 RAM generated from the in-built 1-port memory IP core of the QUARTUS Prime IDE. It is used to hold the user data (pixel data of the original image) inside the processer. The pixel values of the image is loaded to the Data memory using a memory initialization file (.mif) during the compilation process. The content inside instruction memory can be edited during the execution of the program. At each falling edge of the main clock, if the wren is LOW then the output q gives the data stored in the memory location pointed by the input address. If the wren is HIGH, then the input data will be written to the memory location pointed by the input address. Since the memory bits inside the FPGA is used to create the RAM module, the data writing process can be done in single clock cycle.

4. Algorithm

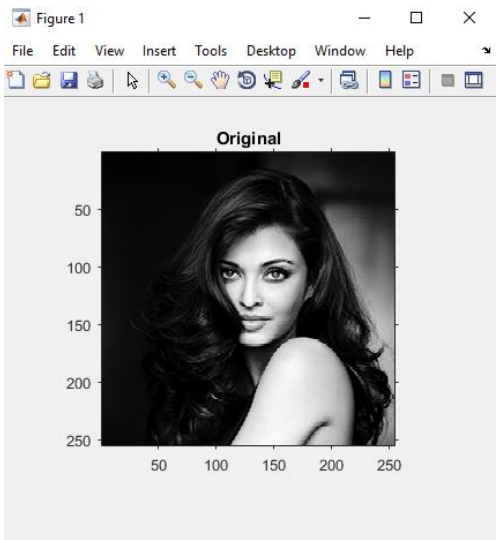
The algorithm of our Verilog design can be divided into several stages which are the loading of pixel values to the registers, gaussian smoothing of the image and the down sampling of the image. Under this algorithm one color plane of an image can be down sampled at a time. So, it was basically implemented on grayscale images. Apart from down sampling an image we designed our custom-made processor with an additional feature of edge detection as well.

The algorithm of the custom-made processor gives the capability to specify image dimensions as $\boldsymbol{a \times b\ pixels}$ where both $\boldsymbol{a, b}$ are odd number less than or equal to 255.

In the following sections the algorithm of down sampling a 255x255 grayscale image is described step by

4.1 Loading the Image

For this explanation let's take the following original 255x255 gray scale image. Using MATLAB this original image is converted into a MIF file which can be directly loaded into the data memory of our design using in-system memory access feature.



In the data memory all the pixel values will be stored like in a stack of height 65025 (ie.255x255) with the pixel indexes as the addresses of each data memory slot containing a pixel value. Indexes of pixels in our original image would be as follows,

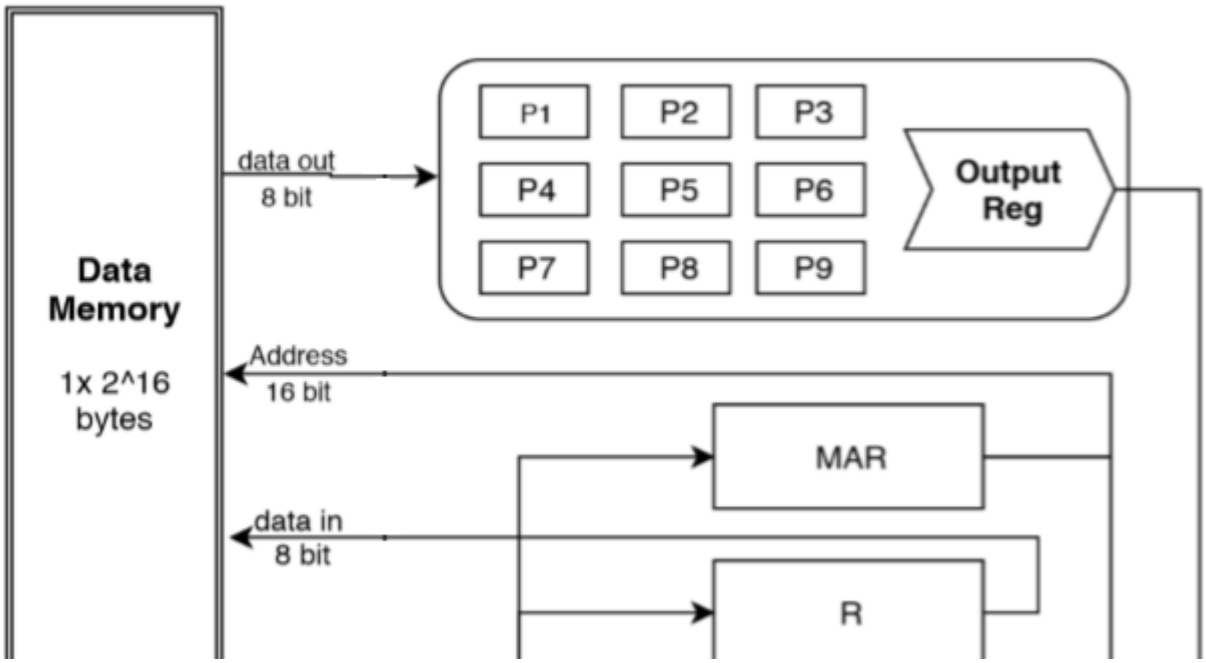
0	1	2	3	4	.....	254
255	256	257	258	259		509
510	511	.....				
.....						

=

4.2 Loading Pixel Values of a 3x3 Kernel into the P Registers of the Processor

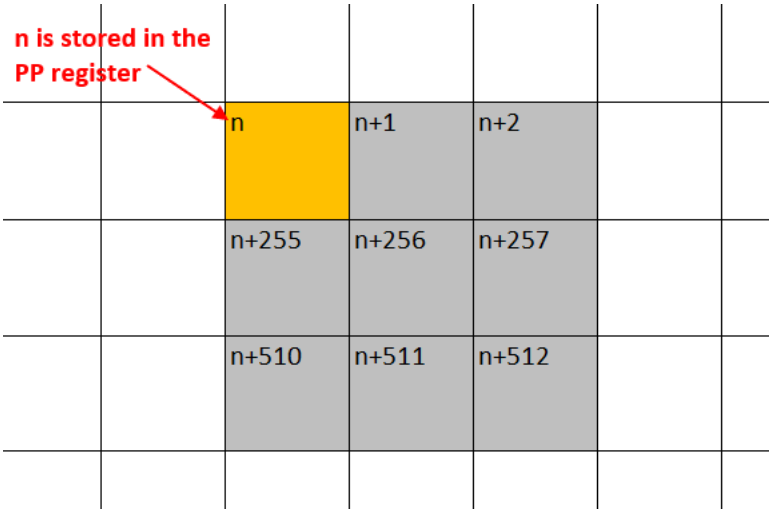
Details about the registers and the assembly instruction set are already explained in the section 2 of this document. So now we'll look into their implementation.

The process of the down sampling starts with the loading of pixel values into P registers. There are 9 such registers P1-P9. Nine pixel values from a 3x3 kernel, placed on the image are loaded in to those 9 registers. Only one P registered is loaded at a time while the index of the relevant pixel being stored in the MAR register. Following extraction of the data path shows how we access our data memory more precisely.



The question which arises with this loading process is how we select the pixels to be loaded in to those 9 P-registers. It is the main part of our algorithm.

9 addresses of the 9 pixels to be loaded into the p-registers in a cycle are put into the MAR register after calculating those relative to a specific pixel index called the **pixel pointer**. The pixel pointer is the pixel index stored in the **PP register** and this is always the index of the top left corner pixel of the current 3x3 image kernel which is being loaded into the P registers. The addresses to be put into the MAR register are calculated in the following manner. 3x3 kernel which is being loaded into the p-registers is colored in grey in the following figure.



4.3 Calculating and Storing

Once nine-pixel values are loaded into the p-registers in one cycle for a particular 3x3 section of the original image we have to do a processing part. This processing part includes gaussian smoothing and down sampling by a factor of 2. Both of these are actually combined in our algorithm.

In the P-register module there is a combinational circuit doing this calculation part which takes 9 inputs from the 9 p-registers and stores the output in the **output reg** which is shown in the above extraction of the data path.

In that those 9-pixel values are firstly multiplied by the weights of the following gaussian kernel used for smoothing the image.

1	2	1
2	4	2
1	2	1

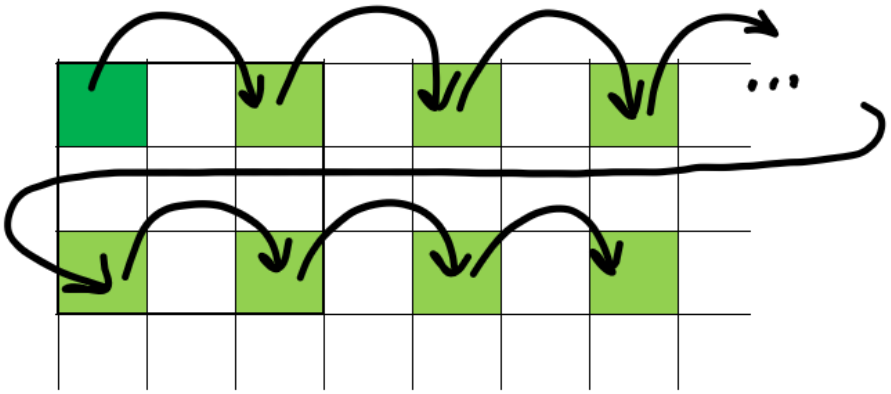
After multiplying with the weights all those values are added together and the result is stored in the output reg. Since the sum of the weights in the gaussian kernel in 16 we have to divide this number stored in the output reg by 16. To do that we neglect the least significant 4 bits of the output reg and take the remaining 8 bits as the final value after diving by 16. That value is stored back in the data memory at the memory location specified in the MAR register.

Likewise, in one cycle 9-pixel values are loaded from a 3x3 section on the original image and one calculated value is stored back into the memory. In the next cycle the same thing will happen to another 3x3 section. For perfect understanding of the algorithm let's look in what these cycles are, how the 3x3 kernel in moved on the image and where the calculated result is stored.

Since the pixel indexes of 9 pixels inside a kernel is calculated relative to the pixel pointer stored in the PP register, the only thing that we have to do in selecting a 3x3 section on the image is specifying the pixel pointer. By updating the pixel pointer, a new 3x3 section get automatically selected.

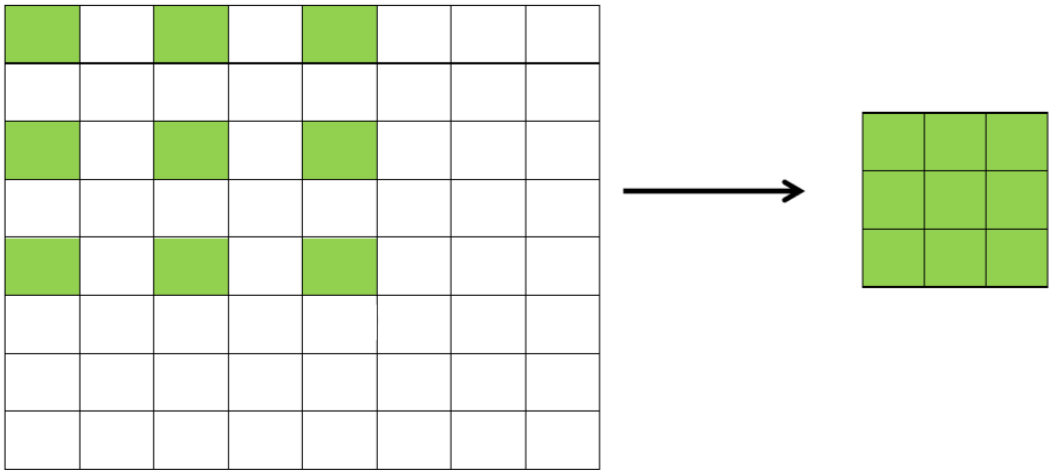
So in after every cycle of loading 9 pixels, calculating the result and storing in the memory; what we do is updating the pixel pointer to repeat the same process for another set of 9 pixels.

Since our down sampling factor is 2, the pixel pointer is updated in the following manner.



If the above diagram is clearly observed it gets convenient that if the green colored pixels are aggregated to make a separate image, it would result in a down sampled version of the original image with a down sampling factor of 2. If we store the calculated result of 9 pixels in a 3x3 section in the pixel pointer relevant to that 3x3 section, which is the green colored pixel according to above diagram, we can achieve our down sampling task.

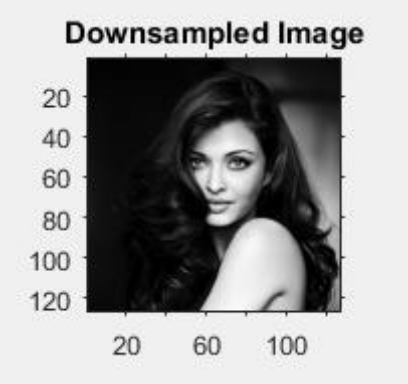
Furthermore, this down sampling method can be illustrated graphically as below.



4.4 Representing the Down Sampled Image

But when storing the calculated values back in the data memory we do not store them in the exact same locations as the green colored ones. Instead the calculated values are stored in the pixels in the index order of 0,1,2,3, ... That means storing is done by sequentially filling the data memory from the beginning. It doesn't matter where we actually stored the calculated valued as long as we graphically represent them correctly. We store the calculated values in that manner just to ease the process of getting the down sampled image out to our computer.

When the image is down sampled by a factor of two we'll end up with a down sampled image of the dimension 127x127. After the process we read the first 16129 pixels (127x127) of the data memory and **export it as a hex file**. That hex file is decoded using a MATLAB code to show the down sampled image as below.



4.5 Assembly Code of implementing the algorithm

Now that we have studied what happens in the algorithm, we can easily understand the following assembly code which was actually used for down sampling a 255x255 grayscale image by a factor of 2. The assembly instruction set is given in the section 2 of this document.

Assembly Code

```
1.  CLH          //Clear H, X, Y
2.  CLX
3.  CLY
4.  SETX 127     //Setting boundary value of the kernel movement in X, Y directions
5.  SETY 127

//Loop begin:

6.  H2PP
7.  H2MAR
8.  LDP1
9.  ADD 1
10. H2MAR
11. LDP2
12. ADD 1
13. H2MAR
14. LDP3
15. ADD 253
16. H2MAR
17. LDP4
18. ADD 1
19. H2MAR
20. LDP5
21. ADD 1
22. H2MAR
23. LDP6
24. ADD 253
25. H2MAR
26. LDP7
27. ADD 1
28. H2MAR
29. LDP8
30. ADD 1
31. H2MAR
32. LDP9

33. CAL          //store the calculated pixel result of the kernel to R register
34. G2MAR        //loading MAR with the memory location to store that calculated value
35. STR          //Store to memory
```

Loading 9-pixel values from the current placement of the 3x3 gaussian kernel in to P1, P2, …, P9 registers of the processor

```
36. STR
37. ING1
38. PP2H
39. ADD 2
40. INX
41. JUMPX 5        //loop in X direction (→)

42. CLX
43. ADD 256
44. INY
45. JUMPY 5        //loop in Y direction (↓)

46. STOP           //Stop process at the end of down sampling
```

This assembly code is written into a text document. A MATLAB code called compiler converts this assembly code into the machine code and generates a MIF file. This MIF file can be loaded into the instruction memory of our processor design using the in-system memory access feature.

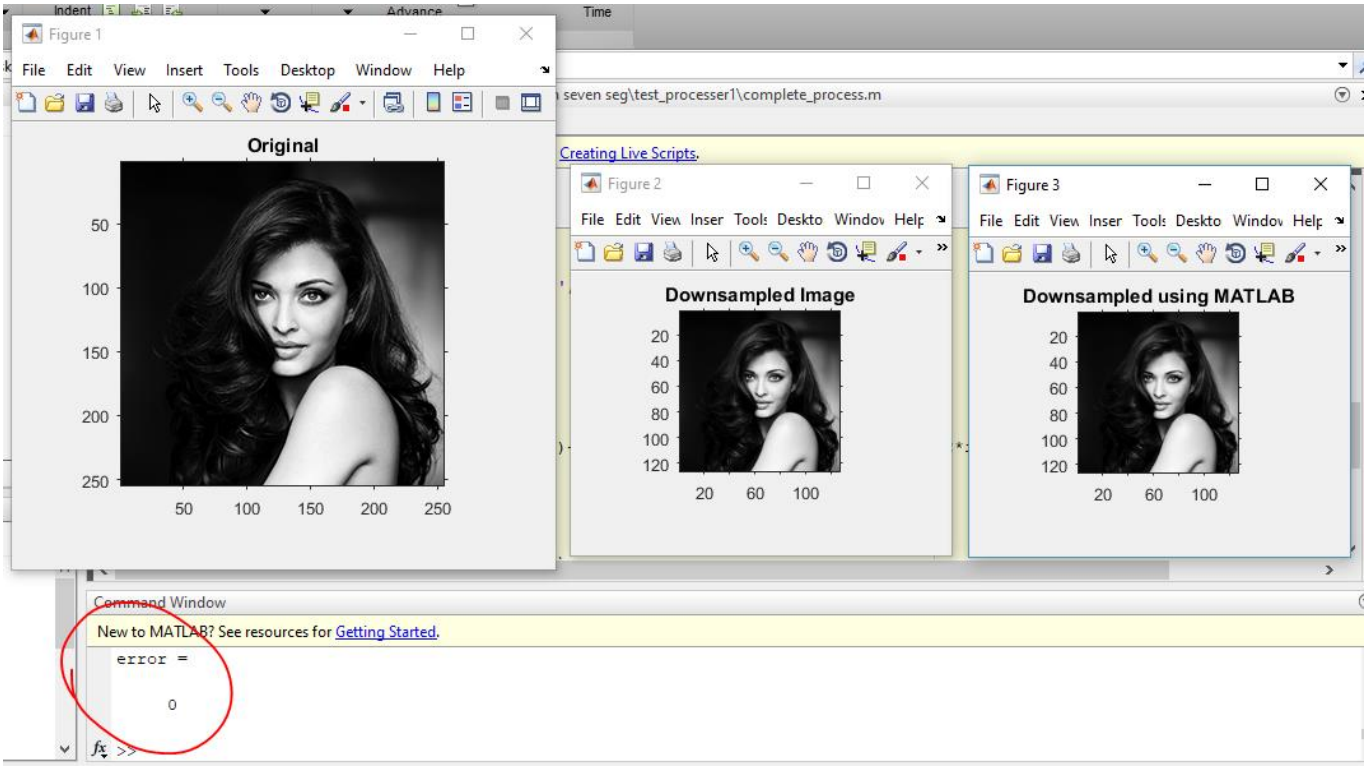
The code can be simply explained like below.

- In the above line 6 to 32 are contains the instructions which would load 9-pixel values into the P registers by using the addresses calculated with reference to the pixel index stored in the PP register.
- In 33, the CAL instruction stores the calculated result relevant to those 9 pixels to the R register.
- Register G is used to keep the index of the pixel where the calculated value in the R register is to be stored back in the data memory. So, in 35,36 the result is stored in the data memory at the address specified in G. Then G is incremented by 1 to point the next address for the next cycle.
- According to 42, the cycles will repeat by the pixel pointer jumping two by two in the x direction. The jumping parameter is stored in register X and it is incremented by 1 in each jump by the instruction INX. In other words when looping in the x direction PP register gets incremented by 2 for each new 3x3 section we meet in x direction. For a 255x255 image there are only 127 such 3x3 sections in x direction. So actually, what happens in 42 is that the jumping happens while X<127.
- Once one row in the image is over it will proceed into the instructions 42-45 which causes the looping in y direction. At the end of this loop we can conclude that the whole image has been successfully down sampled.

5. Testing & Verification

Firstly, the individual Verilog modules were made, and their functionality were separately tested by the input output behavior. Apart from that the functionality of the design was verified at last when the design was completed using MATLAB.

For that we got the down sampled image exported as a hex file as a matrix in MATLAB. We also implemented the exact same down sampling algorithm in MATLAB and down sampled the same original image using MATLAB in the computer and generated another matrix. Then we deducted those 2 matrices and calculated the error between the down sampled image by the custom processor and the down sampled image by the MATLAB code. We got the error as zero.



6. Result Analysis and Verification.

Once the image is passed through the processor to output the down sampled image, analyzing that output for error using a proper technique is important to ensure that the designed processor operates properly. For comparison, an image is down sampled using Matlab and is compared with the down sampled image of the processor for error analysis.

6.1 Generating reference output image.

A Matlab code can be developed to perform the down sampling process and obtain a reference output image which then can be used to compare with the processor output. In the first step, the input image is read as a 255x255 matrix of data type 'double'. A zero array of length 127x127 is made initially to update the final pixel values which are later converted to a 127x127 matrix using the 'reshape' function. Down sampling is done using the average filtering method and the resulting values are updated to the output array. The reshaped array is converted to integer format and displayed. Error is calculated as the difference between the above obtained result and the processor output with the help of the 'sum' function of Matlab. (*Refer Appendix for the Matlab code*).

Stages of the reference output generation is as follows.

Original image – 255x255 pixels

Down sampled image using Matlab – 127x127 pixels

6.2 Result Verification and Analysis.

The two down sampled images cannot be compared just by the view of it. Hence a comparison is performed using the 'sum' function of Matlab to calculate any error for every pixel value. Output of the FPGA is read from the in-system memory as a MIF file and converted to a hex file. It is then read and reshaped to a 127x127 matrix (*Refer Appendix M for Matlab code*). This matrix is compared with the Matlab down sampled image to calculate the error (Refer Appendix M for Matlab code).

Down-sampled image from the FPGA

Down-sampled image from Matlab

6.3 Error Analysis.

Difference between the pixel values of each and every pixel is calculated and summed to obtain the total error. It was found that the error is zero. This means the two down-sampled images are exactly the same concluding that the processor design is operating as expected and meeting all the stated requirements.

```
>> Overall_process
section 1 is DONE!
section 2 is DONE!

error =

     0
```

The main reason behind this zero error can be stated as the down-sampling technique we have used. Under the averaging method, a 3x3 window is used and the center cell is

multiplied by 16, immediate 4 cells to either sides of it by 8 and the remaining 4 cells by 4 and the total is divided by 64. This operation is performed in FPGA by shifting the bit values. Hence an integer value is obtained for every pixel. Even though the original image is converted to 'double' in Matlab and the output is later converted back to 'integer', the rounding-off error probability is negligible due to this averaging approach.

Hence we can conclude that the designed ISA, algorithm and the Processor implementation works accurately meeting all the requirements.

7. Discussion.

- The processor is design in a manner which utilizes the FPGA resources optimally. Three data buses are used to transfer data between registers and the averaging of pixel values is done inside the p register itself rather than going through the ALU which optimizes the resource allocation and minimizes the processing time.
- One drawback of the processor is that there will be some time involved in sending and retrieving data in and out the FPGA. This is due to the communication method we are using here. Instead of the common uart communication protocol, we are using the in-system memory of the FPGA and writing and reading of data is done by uploading and extracting data as MIF files. This approach reduces the processing time to a factor of a second and the errors associated with the general uart communication. Hence even though it involves some time in extracting files, this results in overcoming the drawbacks of the uart method and gives accurate results from the processing.
- The algorithm used here which is an average filtering approach eliminates the requirement of implementing a separate filter like a Gaussian filter meanwhile contributing in getting an error free result. All the multiplications and divisions involved in this filtering process are implemented in the FPGA by shifting bit values so that round off errors does not occur.
- The processor is further capable of performing edge detection of the input image as well. It can be done using the same processor architecture.

Acknowledgement.

We would like to make this an opportunity to thank all those who assisted us in making this project a success. Special thanks go to our lecturer Dr. Jayathu Samarawickrama for the guidance and support provided throughout the project. Also it is with great appreciation we express our heartfelt gratitude towards our colleagues for the support given to us in succeeding with this project. Further we would like to thank the support staff of the laboratory for the support given to us by providing necessary equipment and facilities.

Appendix.

- **Appendix A – Test Processor**

```
module test_processer1 (
 input clock50,
 output clk_led,
 output [7:0]status_led,
 output [15:0]red_led,
 output [20:0]Seven_segs);

// wires connecting modules
wire clk; // main clock used

//data buses
wire [15:0]a_bus;
wire [15:0]b_bus;
wire [15:0]c_bus;

wire [15:0]instruction_bus;
wire [7:0] data_bus;

//input to the register read multiplexer
wire [15:0]mar_out,p_out,pc_out,pp_out,x_out,y_out,r_out,g_out;
// wires for flags
wire  x_flag,y_flag;
// wires for control signals
wire data_write_en;
wire [2:0]b_bus_select;
wire [7:0]c_bus_select;
wire [3:0]p_reg_select;
wire p_reg_write_en;
wire [2:0]alu_select;
wire pc_en;
wire stop_flag;
wire start_flag;
wire flag_wr_en,flag_reg_select;

//wires for seven segements
wire [7:0]seg_input;


// Module Instantiation
```

```
//registers
register_p P_REG(.clk(clk), .data_in(data_bus), .reg_select(p_reg_select),
.write_en(p_reg_write_en),.data_out(p_out[11:0]));

register_g MAR(.clk(clk), .write_en(c_bus_select[6]), .data_in(c_bus),
.data_out(mar_out));

register_pc PC(.clk(clk), .write_en(c_bus_select[5]), .pc_en(pc_en), .data_in(c_bus),
.data_out(pc_out));

register_g PP(.clk(clk), .write_en(c_bus_select[4]), .data_in(c_bus),
.data_out(pp_out));

register_g X(.clk(clk), .write_en(c_bus_select[3]), .data_in(c_bus), .data_out(x_out));

register_g Y(.clk(clk), .write_en(c_bus_select[2]), .data_in(c_bus), .data_out(y_out));

register_g R(.clk(clk), .write_en(c_bus_select[1]), .data_in(c_bus), .data_out(r_out));

register_g H(.clk(clk), .write_en(c_bus_select[0]), .data_in(c_bus), .data_out(a_bus));

//---------------------- update 22/5------------
register_g G(.clk(clk), .write_en(c_bus_select[7]), .data_in(c_bus), .data_out(g_out));
//-----------------------------------------------

//register read multiplexer
reg_mux REG_MUX(.in1({6'b000000,instruction_bus[9:0]}),
            .in2({6'b00000000,p_out[11:4]}),
            .in3(mar_out),
            .in4(pc_out),
            .in5(pp_out),
            .in6(x_out),
            .in7(y_out),
            .in8(g_out),
            .select(b_bus_select),
            .out(b_bus) );

//flags
flag FLAGS(.x_reg(x_out), .y_reg(y_out), .x_flag(x_flag), .y_flag(y_flag),
        .clock(clk), .data_in({6'b000000,instruction_bus[9:0]}) ,.wr_en(flag_wr_en),
        .reg_select(flag_reg_select));

// ALU
alu ALU(.select(alu_select), .A_bus(a_bus), .B_bus(b_bus), .C_bus(c_bus));
```

```
//instruction memory 2 byte x 256 lines
instruction_memory I_MEMORY(.clock(clk), .wren(1'b0), .address(pc_out),
.q(instruction_bus));

//data memory 1 byte x 65536 lines
data_memory D_MEMORY(.clock(clk), .wren(data_write_en), .address(mar_out),
.q(data_bus), .data(r_out));

//instruction decoder
instruction_decoder I_DECODER(.opcode(instruction_bus[15:10]),
                              .x_flag(x_flag),
                              .y_flag(y_flag),
                             //  .start_flag(1'b1),
                              .stop_flag(stop_flag),
                              .data_write_en(data_write_en),
                              .b_bus_select(b_bus_select),
                              .c_bus_select(c_bus_select),
                              .p_reg_select(p_reg_select),
                              .p_reg_write_en(p_reg_write_en),
                              .alu_select(alu_select),
                              .pc_en(pc_en),
                              .flag_reg_select(flag_reg_select),
                              .flag_wr_en(flag_wr_en) );




//Seven segment display
seven_segments seg1(.bin(seg_input),.segments(Seven_segs));

//--------------PLL--------------------------------------

//PLL clockGen(.inclk0(clock50),.c0(clk));

//-------------------------------------------------------

reg [31:0]count=1;
reg r_clk =0;
always @(posedge clock50)begin

  if (count==0)begin
    count<=1;
    r_clk<=~r_clk;
```

```
        end else
        count<=count-1;

    end

    //always @(posedge clock50) begin

    //end

    //assign clk=clock50;
    assign clk=r_clk;

    assign clk_led=clk;
    assign status_led=a_bus[7:0];//pc_out   //a_bus
    assign red_led = g_out;
    assign seg_input=pc_out;

endmodule
```

- **Appendix B – ALU**

```
    module alu (
        input [2:0]select,
        input [15:0]A_bus,
        input [15:0]B_bus,
        output reg [15:0]C_bus
    );

    always @(*) begin
        case (select)
            3'b000:  C_bus = A_bus;
            3'b001:  C_bus = B_bus;
            3'b010:  C_bus = 0;
            3'b011:  C_bus = A_bus + 1;

            3'b100:  C_bus = A_bus + 2;
            3'b101:  C_bus = B_bus + 1;
            3'b110:  C_bus = A_bus + B_bus;
            3'b111:  C_bus = A_bus | B_bus;
        endcase
    end
endmodule // alu
```

- **Appendix C – Flags**

```
module flag (
    input clock,
    input [15:0]x_reg, // Input from the X register
    input [15:0]y_reg, // Input from the Y register
    input [15:0]data_in,  // immediate data from instructions
    input wr_en,reg_select, //write enable and register select
    output x_flag,     // high if X<128
    output y_flag      // high if Y<128
);
reg [15:0] x_comp, y_comp;

always @ (negedge clock) begin
    if (wr_en==1) begin
            if (reg_select==1)
                    y_comp<=data_in;
            else
                    x_comp<=data_in;
    end
end

assign x_flag = (x_reg<x_comp)? 1:0 ; //127 for 256 image 5 for 11 image
assign y_flag = (y_reg<y_comp)? 1:0 ;

endmodule
```

- **Appendix D – Module Register G**

```
module register_g (
    input clk,    // Clock
    input write_en, // Write Enable
    input [15:0]data_in, // Data input from C bus
    output reg [15:0]data_out //Data output to the B bus
);

initial begin
 data_out=0;
end

always @ (negedge clk) begin
    if (write_en) data_out<=data_in;
end
```

```
endmodule
```

- **Appendix E – Instruction Decoder**

```
module instruction_decoder(
    input  [5:0] opcode,
    input  x_flag,y_flag,//start_flag,
    output reg data_write_en,
    output reg [3:0]b_bus_select,
    output reg [7:0]c_bus_select,
    output reg [3:0]p_reg_select,
    output reg p_reg_write_en,
    output reg [2:0]alu_select,
    output reg pc_en,
    output reg stop_flag,
    output reg flag_wr_en,
    output reg flag_reg_select
    );


 parameter NOOP  =6'b000001;
 parameter LDP1  =6'b000010;
 parameter LDP2  =6'b000011;
 parameter LDP3  =6'b000100;
 parameter LDP4  =6'b000101;
 parameter CAL   =6'b000110;
 parameter STR   =6'b000111;
 parameter PP2H  =6'b001000;
 parameter H2PP  =6'b001001;
 parameter ADD   =6'b001010;
 parameter SUB   =6'b001011;
 parameter ING1  =6'b001100;
 parameter G2MAR =6'b001101;
 parameter CLH   =6'b001110;
 parameter CLX   =6'b001111;
 parameter CLY   =6'b010000;
 parameter INX   =6'b010001;
 parameter INY   =6'b010010;
 parameter JUMPX =6'b010011;
 parameter JUMPY =6'b010100;
 parameter H2MAR =6'b010101;
 parameter PP2MAR=6'b010110;
 parameter WRITE =6'b010111;
```

```
parameter STOP  =6'b011000;
parameter LDP5  =6'b011001;
parameter LDP6  =6'b011010;
parameter LDP7  =6'b011011;
parameter LDP8  =6'b011100;
parameter LDP9  =6'b011101;
parameter SETX  =6'b011110;
parameter SETY  =6'b011111;



always @(*) begin

    //if (start_flag) begin
            case (opcode)
                    NOOP: begin
                            data_write_en  <= 1'b0;
                            b_bus_select   <= 3'b000;      //connected to immediate data
                            c_bus_select   <= 8'b00000000;  //no register selected
                            p_reg_select   <= 4'b0000;       //p1 is selected
                            p_reg_write_en <= 1'b0;        //p reg wright disable
                            alu_select          <= 3'b000;      // c_bus<=a_bus
                            pc_en          <= 1'b1;        // pc incremen by 1
                            stop_flag      <= 1'b0;
                            flag_wr_en     <= 1'b0;
                            flag_reg_select<= 1'b0;
                    end

                    LDP1: begin
                            data_write_en  <= 1'b0;
                            b_bus_select   <= 3'b000;
                            c_bus_select   <= 8'b00000000;
                            p_reg_select   <= 4'b0000;
                            p_reg_write_en <= 1'b1;        //p_reg write enabled.
                            alu_select          <= 3'b000;
                            pc_en          <=1'b1;
                            stop_flag      <= 1'b0;
                            flag_wr_en     <= 1'b0;
                            flag_reg_select<= 1'b0;
                    end

                    LDP2:begin
                            data_write_en  <= 1'b0;
                            b_bus_select   <= 3'b000;
```

```
        c_bus_select   <= 8'b00000000;
        p_reg_select   <= 4'b0001;
        p_reg_write_en <= 1'b1;       //p_reg write enabled.
        alu_select          <= 3'b000;
        pc_en         <= 1'b1;
        stop_flag      <= 1'b0;
        flag_wr_en     <= 1'b0;
        flag_reg_select<= 1'b0;
end

LDP3:begin
        data_write_en  <= 1'b0;
        b_bus_select   <= 3'b000;
        c_bus_select   <= 8'b00000000;
        p_reg_select   <= 4'b0010;
        p_reg_write_en <= 1'b1;       //p_reg write enabled.
        alu_select          <= 3'b000;
        pc_en         <= 1'b1;
        stop_flag      <= 1'b0;
        flag_wr_en     <= 1'b0;
        flag_reg_select<= 1'b0;
end

LDP4:begin
        data_write_en  <= 1'b0;
        b_bus_select   <= 3'b000;
        c_bus_select   <= 8'b00000000;
        p_reg_select   <= 4'b0011;
        p_reg_write_en <= 1'b1;       //p_reg write enabled.
        alu_select          <= 3'b000;
        pc_en         <= 1'b1;
        stop_flag      <= 1'b0;
        flag_wr_en     <= 1'b0;
        flag_reg_select<= 1'b0;
end

LDP5:begin
        data_write_en  <= 1'b0;
        b_bus_select   <= 3'b000;
        c_bus_select   <= 8'b00000000;
        p_reg_select   <= 4'b0100;
        p_reg_write_en <= 1'b1;       //p_reg write enabled.
        alu_select          <= 3'b000;
        pc_en         <= 1'b1;
```

```
        stop_flag      <= 1'b0;
        flag_wr_en     <= 1'b0;
        flag_reg_select<= 1'b0;
end

LDP6:begin
        data_write_en  <= 1'b0;
        b_bus_select   <= 3'b000;
        c_bus_select   <= 8'b00000000;
        p_reg_select   <= 4'b0101;
        p_reg_write_en <= 1'b1;        //p_reg write enabled.
        alu_select          <= 3'b000;
        pc_en          <= 1'b1;
        stop_flag      <= 1'b0;
        flag_wr_en     <= 1'b0;
        flag_reg_select<= 1'b0;
end

LDP7:begin
        data_write_en  <= 1'b0;
        b_bus_select   <= 3'b000;
        c_bus_select   <= 8'b00000000;
        p_reg_select   <= 4'b0110;
        p_reg_write_en <= 1'b1;        //p_reg write enabled.
        alu_select          <= 3'b000;
        pc_en          <= 1'b1;
        stop_flag      <= 1'b0;
        flag_wr_en     <= 1'b0;
        flag_reg_select<= 1'b0;
end

LDP8:begin
        data_write_en  <= 1'b0;
        b_bus_select   <= 3'b000;
        c_bus_select   <= 8'b00000000;
        p_reg_select   <= 4'b0111;
        p_reg_write_en <= 1'b1;        //p_reg write enabled.
        alu_select          <= 3'b000;
        pc_en          <= 1'b1;
        stop_flag      <= 1'b0;
        flag_wr_en     <= 1'b0;
        flag_reg_select<= 1'b0;
end
```

```
LDP9:begin
        data_write_en  <= 1'b0;
        b_bus_select   <= 3'b000;
        c_bus_select   <= 8'b00000000;
        p_reg_select   <= 4'b1000;
        p_reg_write_en <= 1'b1;       //p_reg write enabled.
        alu_select          <= 3'b000;
        pc_en         <= 1'b1;
        stop_flag      <= 1'b0;
        flag_wr_en     <= 1'b0;
        flag_reg_select<= 1'b0;
end

CAL:begin
        data_write_en  <= 1'b0;
        b_bus_select   <= 3'b001;
        c_bus_select   <= 8'b00000010;
        p_reg_select   <= 4'b0000;
        p_reg_write_en <= 1'b0;       //p_reg write desabled.
        alu_select          <= 3'b001;      // c_bus<=b_bus
        pc_en         <= 1'b1;
        stop_flag      <= 1'b0;
        flag_wr_en     <= 1'b0;
        flag_reg_select<= 1'b0;
end

STR:begin
        data_write_en  <= 1'b1;       //data memory write is enabled
        b_bus_select   <= 3'b000;
        c_bus_select   <= 8'b00000000;
        p_reg_select   <= 4'b0000;
        p_reg_write_en <= 1'b0;
        alu_select          <= 3'b000;
        pc_en         <= 1'b1;
        stop_flag      <= 1'b0;
        flag_wr_en     <= 1'b0;
        flag_reg_select<= 1'b0;
end

PP2H:begin
        data_write_en  <= 1'b0;
        b_bus_select   <= 3'b100;
        c_bus_select   <= 8'b00000001;
        p_reg_select   <= 4'b0000;
```

```
        p_reg_write_en <= 1'b0;
        alu_select      <= 3'b001;
        pc_en       <= 1'b1;
        stop_flag    <= 1'b0;
        flag_wr_en   <= 1'b0;
        flag_reg_select<= 1'b0;
end

H2PP:begin
        data_write_en  <= 1'b0;
        b_bus_select   <= 3'b000;
        c_bus_select   <= 8'b00010000;
        p_reg_select   <= 4'b0000;
        p_reg_write_en <= 1'b0;
        alu_select      <= 3'b000;
        pc_en       <= 1'b1;
        stop_flag    <= 1'b0;
        flag_wr_en   <= 1'b0;
        flag_reg_select<= 1'b0;
end

ADD:begin
        data_write_en  <= 1'b0;
        b_bus_select   <= 3'b000;
        c_bus_select   <= 8'b00000001;
        p_reg_select   <= 4'b0000;
        p_reg_write_en <= 1'b0;
        alu_select      <= 3'b110;
        pc_en       <= 1'b1;
        stop_flag    <= 1'b0;
        flag_wr_en   <= 1'b0;
        flag_reg_select<= 1'b0;
end

SUB:begin
        data_write_en  <= 1'b0;
        b_bus_select   <= 3'b000;
        c_bus_select   <= 8'b00000001;
        p_reg_select   <= 4'b0000;
        p_reg_write_en <= 1'b0;
        alu_select      <= 3'b111;
        pc_en       <= 1'b1;
        stop_flag    <= 1'b0;
        flag_wr_en   <= 1'b0;
```

```
        flag_reg_select<= 1'b0;
end

ING1:begin
        data_write_en  <= 1'b0;
        b_bus_select   <= 3'b111;
        c_bus_select   <= 8'b10000000;
        p_reg_select   <= 4'b0000;
        p_reg_write_en <= 1'b0;
        alu_select          <= 3'b101;
        pc_en         <= 1'b1;
        stop_flag      <= 1'b0;
        flag_wr_en     <= 1'b0;
        flag_reg_select<= 1'b0;
end

G2MAR:begin
        data_write_en  <= 1'b0;
        b_bus_select   <= 3'b111;
        c_bus_select   <= 8'b01000000;
        p_reg_select   <= 4'b0000;
        p_reg_write_en <= 1'b0;
        alu_select          <= 3'b001;
        pc_en         <= 1'b1;
        stop_flag      <= 1'b0;
        flag_wr_en     <= 1'b0;
        flag_reg_select<= 1'b0;
end

CLH:begin
        data_write_en  <= 1'b0;
        b_bus_select   <= 3'b000;
        c_bus_select   <= 8'b00000001;
        p_reg_select   <= 4'b0000;
        p_reg_write_en <= 1'b0;
        alu_select          <= 3'b010;
        pc_en         <= 1'b1;
        stop_flag      <= 1'b0;
        flag_wr_en     <= 1'b0;
        flag_reg_select<= 1'b0;
end

CLX:begin
        data_write_en  <= 1'b0;
```

```
        b_bus_select   <= 3'b000;
        c_bus_select   <= 8'b00001000;
        p_reg_select   <= 4'b0000;
        p_reg_write_en <= 1'b0;
        alu_select          <= 3'b010;
        pc_en          <= 1'b1;
        stop_flag      <= 1'b0;
        flag_wr_en     <= 1'b0;
        flag_reg_select<= 1'b0;
end

CLY:begin
        data_write_en  <= 1'b0;
        b_bus_select   <= 3'b000;
        c_bus_select   <= 8'b00000100;
        p_reg_select   <= 4'b0000;
        p_reg_write_en <= 1'b0;
        alu_select          <= 3'b010;
        pc_en          <= 1'b1;
        stop_flag      <= 1'b0;
        flag_wr_en     <= 1'b0;
        flag_reg_select<= 1'b0;
end

INX:begin
        data_write_en  <= 1'b0;
        b_bus_select   <= 3'b101;
        c_bus_select   <= 8'b00001000;
        p_reg_select   <= 4'b0000;
        p_reg_write_en <= 1'b0;
        alu_select          <= 3'b101;
        pc_en          <= 1'b1;
        stop_flag      <= 1'b0;
        flag_wr_en     <= 1'b0;
        flag_reg_select<= 1'b0;
end

INY:begin
        data_write_en  <= 1'b0;
        b_bus_select   <= 3'b110;
        c_bus_select   <= 8'b00000100;
        p_reg_select   <= 4'b0000;
        p_reg_write_en <= 1'b0;
        alu_select          <= 3'b101;
```

```
                pc_en         <= 1'b1;
                stop_flag     <= 1'b0;
                flag_wr_en    <= 1'b0;
                flag_reg_select<= 1'b0;
            end

            JUMPX:begin
                if (x_flag) begin
                    data_write_en  <= 1'b0;
                    b_bus_select   <= 3'b000;
                    c_bus_select   <= 8'b00100000;
                    p_reg_select   <= 4'b0000;
                    p_reg_write_en <= 1'b0;
                    alu_select          <= 3'b001;
                    pc_en         <= 1'b1;
                    stop_flag     <= 1'b0;
                    flag_wr_en    <= 1'b0;
                    flag_reg_select<= 1'b0;
                end else begin
                    data_write_en  <= 1'b0;
                    b_bus_select   <= 3'b000;       //connected to immediate
data
                    c_bus_select   <= 8'b00000000;  //no register selected
                    p_reg_select   <= 4'b0000;      //p1 is selected
                    p_reg_write_en <= 1'b0;        //p reg wright disable
                    alu_select          <= 3'b000;     // c_bus<=a_bus
                    pc_en         <= 1'b1;        // pc incremen by 1
                    stop_flag     <= 1'b0;
                    flag_wr_en    <= 1'b0;
                    flag_reg_select<= 1'b0;
                end
            end

            JUMPY:begin
                if (y_flag) begin
                    data_write_en  <= 1'b0;
                    b_bus_select   <= 3'b000;
                    c_bus_select   <= 8'b00100000;
                    p_reg_select   <= 4'b0000;
                    p_reg_write_en <= 1'b0;
                    alu_select          <= 3'b001;
                    pc_en         <= 1'b1;
                    stop_flag     <= 1'b0;
                    flag_wr_en    <= 1'b0;
```

```
                flag_reg_select<= 1'b0;
            end else begin
                data_write_en  <= 1'b0;
                b_bus_select   <= 3'b000;       //connected to immediate
data
                c_bus_select   <= 8'b00000000;  //no register selected
                p_reg_select   <= 4'b0000;      //p1 is selected
                p_reg_write_en <= 1'b0;        //p reg wright disable
                alu_select          <= 3'b000;      // c_bus<=a_bus
                pc_en          <= 1'b1;        // pc incremen by 1
                stop_flag      <= 1'b0;
                flag_wr_en     <= 1'b0;
                flag_reg_select<= 1'b0;
            end
        end

        H2MAR:begin
            data_write_en  <= 1'b0;
            b_bus_select   <= 3'b000;
            c_bus_select   <= 8'b01000000;
            p_reg_select   <= 4'b0000;
            p_reg_write_en <= 1'b0;
            alu_select          <= 3'b000;
            pc_en          <= 1'b1;
            stop_flag      <= 1'b0;
            flag_wr_en     <= 1'b0;
            flag_reg_select<= 1'b0;
        end

        PP2MAR:begin
            data_write_en  <= 1'b0;
            b_bus_select   <= 3'b100;
            c_bus_select   <= 8'b01000000;
            p_reg_select   <= 4'b0000;
            p_reg_write_en <= 1'b0;
            alu_select          <= 3'b001;
            pc_en          <= 1'b1;
            stop_flag      <= 1'b0;
            flag_wr_en     <= 1'b0;
            flag_reg_select<= 1'b0;
        end

        STOP:begin
            data_write_en  <= 1'b0;
```

```
        b_bus_select   <= 3'b000;      //connected to immediate data
        c_bus_select   <= 8'b00000000;  //no register selected
        p_reg_select   <= 4'b0000;      //p1 is selected
        p_reg_write_en <= 1'b0;        //p reg wright disable
        alu_select          <= 3'b000;      // c_bus<=a_bus
        pc_en          <= 1'b0;        // pc is desabled
        stop_flag      <= 1'b1;
        flag_wr_en     <= 1'b0;
        flag_reg_select<= 1'b0;
end

SETX: begin
        data_write_en  <= 1'b0;
        b_bus_select   <= 3'b000;      //connected to immediate data
        c_bus_select   <= 8'b00000000;  //no register selected
        p_reg_select   <= 4'b0000;      //p1 is selected
        p_reg_write_en <= 1'b0;        //p reg wright disable
        alu_select          <= 3'b000;      // c_bus<=a_bus
        pc_en          <= 1'b1;        // pc incremen by 1
        stop_flag      <= 1'b0;
        flag_wr_en     <= 1'b1;
        flag_reg_select<= 1'b0;
end

SETY: begin
        data_write_en  <= 1'b0;
        b_bus_select   <= 3'b000;      //connected to immediate data
        c_bus_select   <= 8'b00000000;  //no register selected
        p_reg_select   <= 4'b0000;      //p1 is selected
        p_reg_write_en <= 1'b0;        //p reg wright disable
        alu_select          <= 3'b000;      // c_bus<=a_bus
        pc_en          <= 1'b1;        // pc incremen by 1
        stop_flag      <= 1'b0;
        flag_wr_en     <= 1'b1;
        flag_reg_select<= 1'b1;
end

default: begin
        data_write_en  <= 1'b0;
        b_bus_select   <= 3'b000;      //connected to immediate data
        c_bus_select   <= 8'b00000000;  //no register selected
        p_reg_select   <= 4'b0000;      //p1 is selected
        p_reg_write_en <= 1'b0;        //p reg wright disable
        alu_select          <= 3'b000;      // c_bus<=a_bus
```

```
                        pc_en         <= 1'b1;        // pc incremen by 1
                        stop_flag     <= 1'b0;
                        flag_wr_en    <= 1'b0;
                        flag_reg_select<= 1'b0;
                    end

            endcase
        //end
    end


endmodule // instruction_decoder
```

- **Appendix F – Register PC**

```
module register_pc (
    input clk,    // Clock
    input write_en, // Write Enable
    input pc_en, // hold the current pc (this is for micro instructions)
    input [15:0]data_in, // Data input from C bus
    output reg [15:0]data_out //Data output to the B bus
);

initial begin
 data_out=0;
end

always @ (negedge clk) begin
    if (pc_en) begin
            if (write_en) data_out<=data_in;
            else data_out<=data_out+1;
    end
end

endmodule
```

- **Appendix G – Register P**

```
module register_p (
    input clk,    // Clock
    input [7:0]data_in, // input from the dta memory bus
    input [3:0]reg_select, // select the loading registry
    input write_en,  // write enable
    output [11:0]data_out // output to the B bus
```

```
);

reg [7:0] p1,p2,p3,p4,p5,p6,p7,p8,p9;

initial begin
    p1<=0;
    p2<=0;
    p3<=0;
    p4<=0;
    p5<=0;
    p6<=0;
    p7<=0;
    p8<=0;
    p9<=0;
end

always @(negedge clk) begin
    if (write_en) begin
        case (reg_select)
            4'b0000: p1<=data_in;
            4'b0001: p2<=data_in;
            4'b0010: p3<=data_in;
            4'b0011: p4<=data_in;
            4'b0100: p5<=data_in;
            4'b0101: p6<=data_in;
            4'b0110: p7<=data_in;
            4'b0111: p8<=data_in;
            4'b1000: p9<=data_in;

        endcase
    end
end

assign
data_out=(p1+{p2,1'b0}+p3+{p4,1'b0}+{p5,2'b00}+{p6,1'b0}+p7+{p8,1'b0}+p9);

endmodule
```

- **Appendix H – Register mux**

```
module reg_mux(
input [15:0]in1,in2,in3,in4,in5,in6,in7,in8,
output [15:0]out,
input [2:0]select
```

```
    );

    assign out   = (select==3'b000)?  in1
                         : (select==3'b001)?  in2
                         : (select==3'b010)?  in3
                         : (select==3'b011)?  in4
                         : (select==3'b100)?  in5
                         : (select==3'b101)?  in6
                         : (select==3'b110)?  in7
                                                  :  in8;
    endmodule // reg_mux
```

- **Appendix I – Module Seven Segments**

```
module seven_segments(bin,segments);

        input [7:0] bin;

        output [20:0] segments;

        wire [11:0]bcd_out;

        bin2bcd dec(.bin(bin),.bcd(bcd_out));

        segment7 s1(.bcd(bcd_out[3:0]),.seg(segments[6:0]));

        segment7 s2(.bcd(bcd_out[7:4]),.seg(segments[13:7]));

        segment7 s3(.bcd(bcd_out[11:8]),.seg(segments[20:14]));

endmodule



module segment7(
    bcd,
    seg
   );

    //Declare inputs,outputs and internal variables.
    input [3:0] bcd;
    output [6:0] seg;
```

```
    reg [6:0] seg;
//always block for converting bcd digit into 7 segment format
   always @(bcd)
   begin
      case (bcd) //case statement
         0 : seg = 7'b0000001;
         1 : seg = 7'b1001111;
         2 : seg = 7'b0010010;
         3 : seg = 7'b0000110;
         4 : seg = 7'b1001100;
         5 : seg = 7'b0100100;
         6 : seg = 7'b0100000;
         7 : seg = 7'b0001111;
         8 : seg = 7'b0000000;
         9 : seg = 7'b0000100;
         //switch off 7 segment character when the bcd digit is not a decimal number.
         default : seg = 7'b1111111;
      endcase
      //seg=~seg;

   end
endmodule



module bin2bcd(
   bin,
    bcd
```

```
);
//input ports and their sizes
input [7:0] bin;
//output ports and, their size
output [11:0] bcd;
//Internal variables
reg [11 : 0] bcd;
 reg [3:0] i;

 //Always block - implement the Double Dabble algorithm
 always @(bin)
   begin
      bcd = 0; //initialize bcd to zero.
      for (i = 0; i < 8; i = i+1) //run for 8 iterations
      begin
         bcd = {bcd[10:0],bin[7-i]}; //concatenation

         //if a hex digit of 'bcd' is more than 4, add 3 to it.
         if(i < 7 && bcd[3:0] > 4)
            bcd[3:0] = bcd[3:0] + 3;
         if(i < 7 && bcd[7:4] > 4)
            bcd[7:4] = bcd[7:4] + 3;
         if(i < 7 && bcd[11:8] > 4)
            bcd[11:8] = bcd[11:8] + 3;
      end
   end
endmodule
```

- **Appendix J – Image to MIF conversion – Matlab**

```
% run this section first. this will convert the original
% image file to a.mif file.
clear all; close all;

No_of_lines=255^2;
img=imread('google2_255.png');
img_o=img(:,:,1);
figure;
imshow(img_o)
output=reshape(img_o,[1,No_of_lines]);

%Seperate the RGB colors to 3 seperate images:


%create mif file
fid = fopen(strcat('Image','.mif'),'w');
str = 'WIDTH=8;\nDEPTH=65536;\n\nADDRESS_RADIX=UNS;\nDATA_RADIX=UNS;\n\n';
fprintf(fid,str);
str = 'CONTENT BEGIN';
fprintf(fid,'%s\n',str);

for k=1:No_of_lines
    fprintf(fid,'%d : %d;\n',k-1,output(k));
end
fprintf(fid,'END;\n');
fclose(fid);
disp('section 1 is DONE!');
```

- **Appendix K – Hex file to image conversion – Matlab**

```
% run ths section after you successfuly exported tha data
% from in-system memory content editor to a .hex file.
%this section will convert the .hex file to an image file.
clear all
im_length=127^2;
im_data=zeros(1,im_length);
fo=fopen('downsampled_image.hex','r');
for i=1:im_length
   line=fgetl(fo);
   data_hex=line(10:11);
   data_dec=(sscanf(data_hex,'%x'));
   im_data(i)=data_dec;
end
fclose(fo);
image=reshape(im_data,127,127);
image=uint8(image);
figure;
imshow(image)
disp('section 2 is DONE!');
```

- **Appendix L – Compiler – Matlab**

```
clear all; close all;
```

```
program=fopen('data_and.txt','r');
B=fscanf(program,'%c',inf);
fclose(program);
C = strsplit(B,'\n');
No_of_lines=length(C);

out_file=fopen('instruction_memory.txt','w');
output=zeros(1,No_of_lines);
for i=1:No_of_lines
    D = strsplit(char(C(i)));
    size= length(D);
    switch char(D(1))
        case 'NOOP'
            output(i)=1024;
        case 'LDP1'
            output(i)=2048;
        case 'LDP2'
            output(i)=3072;
        case 'LDP3'
            output(i)=4096;
        case 'LDP4'
            output(i)=5120;
        %%-------edit for gaussian
        case 'LDP5'
            output(i)=25600;
        case 'LDP6'
            output(i)=26624;
        case 'LDP7'
            output(i)=27648;
        case 'LDP8'
            output(i)=28672;
        case 'LDP9'
            output(i)=29696;
        %%--------
        case 'CAL'
            output(i)=6144;
        case 'STR'
            output(i)=7168;
        case 'PP2H'
            output(i)=8192;
        case 'H2PP'
            output(i)=9216;
        case 'ADD'
            output(i)=10240+str2double(char(D(2)));
        case 'SUB'
            output(i)=11264+str2double(char(D(2)));
        case 'ING1'
            output(i)=12288;
        case 'G2MAR'
            output(i)=13312;
        case 'CLH'
            output(i)=14336;
        case 'CLX'
            output(i)=15360;
        case 'CLY'
            output(i)=16384;
        case 'INX'
            output(i)=17408;
        case 'INY'
            output(i)=18432;
        case 'JUMPX'
```

```
            output(i)=19456+str2double(char(D(2)));
        case 'JUMPY'
            output(i)=20480+str2double(char(D(2)));
        case 'H2MAR'
            output(i)=21504;
        case 'PP2MAR'
            output(i)=22528;
        case 'SETX'
            output(i)=30720+str2double(char(D(2)));
        case 'SETY'
            output(i)=31744+str2double(char(D(2)));
        case 'STOP'
            output(i)=24576;
        otherwise
            output(i)=1024;
    end
    fprintf(out_file,'%d\n',output(i));
end

fclose(out_file);
disp('done!');
No_of_lines

%%create mif file
fid = fopen(('instructions_and.mif'),'w');
str = 'WIDTH=16;\nDEPTH=256;\n\nADDRESS_RADIX=UNS;\nDATA_RADIX=UNS;\n\n';
fprintf(fid,str);
str = 'CONTENT BEGIN';
fprintf(fid,'%s\n',str);
%n = 0;
for k=1:No_of_lines
    fprintf(fid,'%d : %d;\n',k-1,output(k));
   % n=n+1;
end
fprintf(fid,'END;\n');
fclose(fid);
```

- **Appendix M – Error Analysis – Matlab**

```
im =double(imread('google2_255.png'));
% im=ones(255,255)*100;
out=zeros(1,127*127);
index=1;
for i=1:2:253
    for j=1:2:253


out(index)=(im(i,j)+2*im(i,j+1)+im(i,j+2)+2*im(i+1,j)+4*im(i+1,j+1)+2*im(i+
1,j+2)+im(i+2,j)+2*im(i+2,j+1)+im(i+2,j+2))/16;
        index=index+1;

    end
end
 a=reshape(out,127,127);
 a=transpose(uint8(a));
 figure;
```

```
imshow(a);
error=sum(sum(image-a))
```